Introducción al lenguaje de programación Python

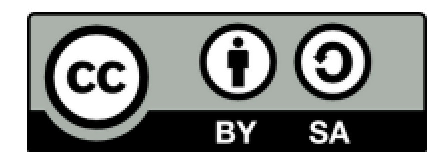


(3ª Edición)

Manual de apoyo al alumno

Oficina de Software Libre de la Universidad de Granada

Centro de Enseñanzas Virtuales de la Universidad de Granada



Advertencia:

Este manual ha sido concebido como apoyo al curso "Introducción al lenguaje de programación Python 3ª Edición" del Centro de Enseñanzas Virtuales de la Universidad de Granada (CEVUG).

Aunque contiene la mayor parte del contenido textual de dicho curso, no está previsto para ser usado independientemente a este, por lo que no debe considerarse ni utilizarse como un texto exhaustivo o completo en sí mismo, si no como una ayuda.

Índice

Introducción a Python. Conceptos generales, instalación, mi primer programa en Python

¿Qué es el software libre?

Aunque en español la palabra libre no tiene tantas connotaciones como la inglesa free, lo cierto es que hablar de tiene cierta ambigüedad: hablamos de software libre como hablamos del sol cuando amanece, no como cuando hablamos de que hay barra libre ni buffet libre. La idea del software libre, la principal, eso que sea libre, no que sea gratis. Por eso, la definición canónica de software libre incluye cuatro libertades

1. La libertad de usar el software de cualquier modo que consideremos razonable.
2. La libertad de estudiar como funciona el programa, para lo cual hay que tener acceso al código fuente.
3. La libertad de redistribuir copias como uno considere conveniente. Lo que implica el poder acceder gratis a esas copias; sin embargo, en sentido estricto, esto es un derecho, no una obligación, por lo que quien adquiere software libre no tiene por qué hacer disponible para todo el mundo esas copias. Sin embargo quien libera el software, como es natural, debe hacerlo.
4. La libertad de distribuir copias modificadas, para lo cual es condición necesaria la segunda libertad.

Ciertamente una de las libertades más importantes es la segunda, por eso se identifica el software libre con el software de fuentes abiertas. En realidad, la diferencia es sólo de énfasis (el SL hace énfasis en la libertad, el de fuentes abiertas en las buenas prácticas asociadas al desarrollo con fuentes abiertas), pero en la práctica son casi lo mismo y lo utilizaremos de forma indistinta dentro de este curso.

Por otro lado, la creación de software libre tiene una importancia vital dentro del ambiente universitario; la investigación adquiere mucha más difusión si se acompaña la publicación de una librería o aplicación de software libre, un alumno que libere sus prácticas tiene más posibilidades de obtener buen resultado con las mismas, o si libera su proyecto fin de carrera es más posible que una persona que quiera darle empleo lo vea y lo evalúe positivamente (aparte del hecho de que puede conseguir mucha más repercusión en el mismo), y finalmente el software creado para la universidad, si se libera, tiene más posibilidades de conseguir aportaciones de la comunidad y de tener un impacto en el resto de las universidades, y permite que la labor de los servicios de informática (que son quienes principalmente lo desarrollan) se divulgue dentro de la comunidad.

Conceptos generales

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90 para administración de sistemas operativos, en concreto de uno olvidado denominado Amoeba.

Como tal es un lenguaje interpretado, dirigido a objetos, y *limpio* por usar en su sintaxis el mínimo de caracteres innecesarios. Por eso mismo es un lenguaje *claro* muy adecuado como primer lenguaje de programación, aunque es actualmente un lenguaje maduro que se puede usar para todo tipo de aplicaciones, desde aplicaciones web hasta temas relacionado con la bioinformática

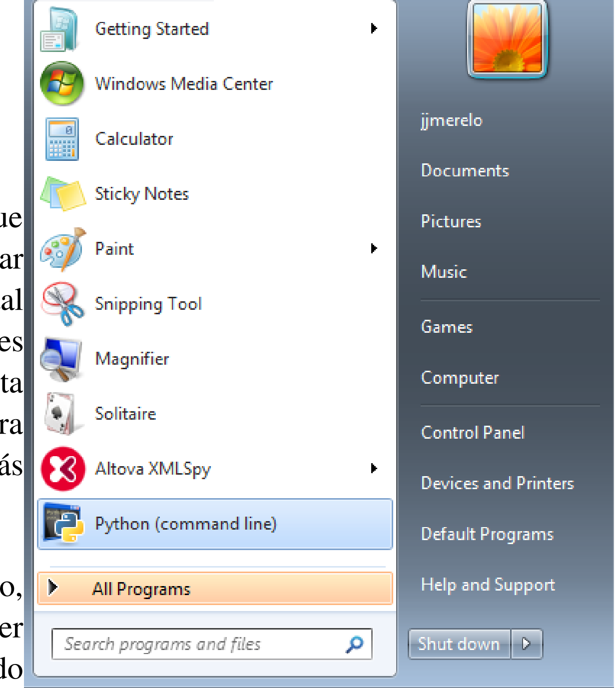
Por otro lado, se trata de un lenguaje de *tipado dinámico*, es decir, que los tipos se asignan a las variables durante el uso de las mismas, y no durante la escritura o compilación de los programas. Por eso es muy fácil trabajar con él, y crear programas flexibles, que se modifiquen sobre la marcha

Hoy en día se le conoce tanto por su patrocinio por parte de Google, que contrató a Guido van Rossum, como por su uso dentro del Google AppSpot o del entorno de aplicaciones Django, de gran aplicación para la creación de portales y aplicaciones basadas en la web. También se usa como lenguaje de creación de aplicaciones en móviles, o para dispositivos empotrados.

En resumen, un lenguaje ameno, fácil de aprender, pero también potente y expresivo. Esperamos que os resulte útil.

Instalación de Python

Introducción y advertencia



Python es un lenguaje interpretado, y como tal, lo que se instala es una aplicación que necesitarás cargar cada vez que quieras ejecutar un programa en tal lenguaje. **No es un entorno de programación**; es decir, no es un entorno completo que te permita editar, ejecutar, depurar el programa y demás. Para eso existen diferentes opciones que veremos más adelante.

Con esto lo que quiero decir es que, una vez instalado, al pinchar en el icono que diga "Python" no va a hacer nada salvo sacar una línea de órdenes (dependiendo del sistema operativo); sí se ejecutará el intérprete automáticamente cuando se pulse el ratón sobre un programa escrito en Python (con la extensión `.py`), pero lo más probable es que, tratándose de programas que se ejecutan en el intérprete de órdenes, no se vea nada. Concluyendo: una vez instalado no te lances a ejecutar nada, espera a

escribir tu primer programa en Python y entonces comprobarás si funciona todo correctamente (que debería hacerlo)

¿Qué versión debería instalar?

En la página de descarga de Python hay dos versiones principales: la 2.7 y la 3.1. Este curso está hecho principalmente con la 2.6, así que la que más se le acerca es la 2.7. Puedes instalar cualquiera de ellas, o las dos, pero aconsejamos para evitar diferencias que se instale la 2.7. En todo caso, actualmente es la versión más usada.

Otra opción que te puedes instalar en un dispositivo portátil para usar en diferentes ordenadores es la Portable Python. En este caso las versiones disponibles están basadas en la 2.6 o 3.0; aconsejamos que se instale la primera

¿Cómo instalo Python en mi ordenador?

Si usas Linux ya sabes como hacerlo; usa tu programa favorito para descargártelo de los repositorios e instalarlo; es posible, de todas formas, que ya esté instalado en la distribución base; Ubuntu, Fedora, Guadalinex y otras ya lo incluyen por ser el lenguaje con el que se escriben muchas de las aplicaciones base que llevan. Para ver si estás instalado abre un intérprete de órdenes (que también es posible que sepas como hacer, pero si no Aplicaciones -> Accesorios -> Terminal) y escribe `python --version` que debería devolver algo así `Python 2.6.5` Si no es así, tendrás que instalarlo.

En Windows o Macintosh hay que descargar alguna de las opciones que se presentan en la página de descargas, teniendo en cuenta las observaciones hechas anteriormente.

En Windows se sigue el proceso habitual en las instalaciones: ir haciendo click sobre "Next" hasta que se indique que se ha instalado. El sistema operativo te hará diferentes advertencias sobre que estás instalando un programa, a las que contestarás que sí, que por supuesto, que faltaría más. Si contestas que no, no te lo va a instalar, así que tendrás que contestar que sí.

Lo que instalarás en Windows será un intérprete de línea de comandos, que aparecerá en el menú como se muestra en la ilustración adjunta, con el texto Python (command line); es decir, Python (línea de órdenes).

C:\Python27\python.exe

```
Python 2.7 (r27:82525, Jul  4 2010, 09:01:59) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

El resultado de la ejecución aparece arriba. Desde ahí se pueden ejecutar directamente órdenes en Python, como veremos más adelante. También se instala un fichero de ayuda al que se puede acceder desde el menú -> Todos los programas -> Python 2.7 -> Python Manuals (o Manuales de Python).

El paquete indicado anteriormente instala también un mini-entorno de desarrollo llamado IDLE, que se puede ejecutar desde el menú anterior eligiendo IDLE. Es un pequeño envoltorio de la línea de comandos, pero te permite cargar ficheros y salvarlos, al menos. Aunque no permite editar programas, sí permite probar órdenes y tiene alguna facilidad adicional como la expansión del nombre de las órdenes (pulsando tabulador). Se puede usar para hacer pruebas, o cuando se conozca más el lenguaje, depurar programas

En cuanto a otras plataformas, este wikilibro describe como hacerlo en un Mac; lo más probable es que esté ya instalado (y que la versión sea la correcta), pero si no puedes instalártelo desde la página de descargas indicada anteriormente

Finalmente, y como curiosidad, puedes instalártelo dentro del "Android Scripting Environment" en tu teléfono Android, tal como los HTC que tan célebres se han hecho últimamente. Así puedes hacer los ejercicios que te planteemos en cualquier lugar, incluso en el bus.

Otros programas

Como se ha dicho más arriba, la instalación de Python proporciona sólo el intérprete que permite ejecutar los programas escritos en él; pero para escribir estos programar hará falta un editor. Todos los sistemas operativos tienen un editor de textos básico, pero se es más productivo usando editores de textos específicos de programadores. En Linux se puede usar emacs, vi, kate, Geany o cualquiera de los múltiples programas que existen, en el Mac hay también buenas opciones

En Windows se puede descargar Geany de su página web, e instalarse con las opciones básicas. Al final de la instalación te ofrece la posibilidad de ejecutarlo.

La ventaja que tiene este tipo de programa es que te colorea las palabras clave y te detecta algunos errores de sintaxis básicos, permitiendo que uno sea más productivo

Para usuarios de entornos de desarrollo tales como NetBeans o Eclipse, hay un plugin denominado PyDev que se puede instalar para editar este tipo de programas. Se puede instalar desde el menú correspondiente, o descargándoselo de la página correspondiente. En esta página instruyen sobre cómo instalárselo en Eclipse, por ejemplo.

Versiones de Python

En el capítulo anterior has visto que, para seguir este curso, se recomienda Python 2.X pese a que, si buscas por internet, verás que existe una versión 3.x

Hoy por hoy, existen dos líneas de Python. Una de ellas es la 2, que es a la que se refiere este curso, y la otra es la 3.

Pese a lo que pudiera parecer, Python 3 **no** es una versión más moderna del mismo Python 2, si no que se trata de una versión distinta y **no compatible** con este.

Las diferencias, en realidad, no son muy grandes, pero sí lo suficiente para que no sean mutuamente compatibles. Un programa escrito para una de las versiones no funcionará (o lo hará mal) bajo el otro intérprete.

Por ejemplo, la línea que en Python 2 se escribe así:

```
print "Hola Mundo"
```

en Python 3 se escribirá así:

```
print ("Hola Mundo")
```

Existen otras muchas diferencias, que pueden verse en la página What’s New In Python 3.0.

La mayoría del software más popular que hoy hoy día existe está escrito con la versión 2, del mismo modo que muchas de las librerías sólo existen para esta versión, o tienen versiones menos estable o actualizadas para la versión 3.

Además, conociendo Python 2 es muy fácil ver las diferencias con python 3 y aplicarlas.

Por todo esto, la mayoría de los cursos, tutoriales y libros se escriben para la versión 2, y este no es una excepción.

Mi primer programa en Python

El primer programa suele ser el clasico que imprime un mensaje por la pantalla. En Python sería de esta forma

```
print "Hola, mundo libre"
```

Que aparecerá de la forma siguiente si lo ejecutamos directamente desde el intérprete

```
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56) [GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information. >>>
print "Hola, mundo libre" Hola, mundo libre >>>
```

Como veis, tiene dos partes: la orden `print` y la cadena de caracteres, entre comillas, `"Hola, mundo libre"`. A diferencia de otros lenguajes, en muchos casos en Python se pueden evitar los paréntesis en las llamadas a funciones; este es uno de ellos. Resulta obvio que el argumento para la función `print` es la cadena, así que ¿para qué más?. Por otro lado, la cadena está entre comillas, como resulta habitual, y `print` incluye un retorno de carro al final de la misma, también como suele ser habitual. Por último, se utiliza como terminador de la orden el simple retorno de carro; no hace falta ningún otro carácter para indicarlo. Si ya está claro que se ha acabado, ¿para qué poner más cosas

Lo más habitual en los programas es tenerlos en un fichero aparte. Editémoslo (con alguno de los editores anteriores) y guardémoslo en hola.py. Ya que tenemos abierta la línea de comandos con Python, salgamos de ella (quit() o control-d) y ejecutémoslo directamente con

```
python hola.py
```

El resultado será el mismo de antes, como es natural

En entornos Linux/Unix es habitual también incluir al principio del fichero con el programa una línea que permite ejecutarlo directamente desde línea de comandos, sin necesidad de llamar explícitamente al intérprete. Esta línea, llamada "shebang", suele ser `#!/usr/bin/python` o `#!/usr/bin/env python` (esta última más genérica). El fichero quedaría algo así

```
#!/usr/bin/python
print "Hola, mundo libre"
```

(Nótese la línea en blanco). Para hacerlo ejecutable se sigue también el procedimiento habitual el Linux

```
chmod +x hola.py
```

con lo que posteriormente ya se puede ejecutar también de la forma habitual

```
bash% ./hola.py
```

El "./" puede que no sean necesarios si se tiene el propio directorio incluido en el camino de ejecución, pero no suele ser una práctica habitual (ni segura) así que es mejor curarse en salud haciendo de la forma anterior. El resultado, de todas formas, tiene que ser el mismo.

UTF-8

Si se tienen problemas para introducir ciertos caracteres, como los acentos o la letra Ñ, el interprete de python responderá con un error con un aspecto similar a esto:

```
SyntaxError: Non-ASCII character '\xxx' in file programa.py on line X, but
no encoding declared; see http://www.python.org/peps/pep-0263.html for
details
```

Python, en principio, asume que la codificación es ASCII y, por tanto, no reconocerá caracteres fuera de esa codificación. Para solucionar esto, los documentos deberán ser definidos como UTF-8

Para advertir a python de que este es el formato, la forma más simple es indicarlo por medio de una "Línea mágica". Justo debajo del shebang (eso de #!...), se coloca lo siguiente:

```
# -*- coding: utf-8 -*-
```

de modo que la cabecera quedaría más o menos así:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

Con esto, el interprete de python debería admitir el uso de caracteres UTF-8 en nuestros scripts. En cualquier caso, no es recomendable el uso de caracteres no ASCII en nombre de variables o funciones, etc.

Para más detalles o casos más complejos, se recomienda consultar la página de información que indica el propio mensaje de error.

Estructuras de datos básicas

¿Qué es una variable?

En Informática, una variable es "algo" en lo que puedes almacenar información para su uso posterior.

En Python, una variable puede almacenar un número, una letra, un conjunto de números o de letras o incluso conjuntos de conjuntos.

¿Definir una variable?

Las variables en Python se crean cuando se definen, es decir, cuando se les asigna un valor.

Para crear una variable, tienes que escribir la variable en el lado izquierda , ponemos el signo igual y a continuacion el valor que quieras darle a la derecha.

Vamos a ver ejemplos de definiciones de variables:

- y = 2.5
- esquina = 10 + 5
- Nombre1 = "El alumno ya sabe Python"
- _y2 = 'p'
- dias_de_la_semana = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
- Fecha_de_nacimiento_del curso = ['Lunes', 1, 'Octubre', 2010]
- año = 2010

El nombre de una variable debe empezar por una letra o por un carácter subrayado (_) y puede seguir con más letras, numéros o subrayados.

Puedes utilizar mayúsculas, pero ten en cuenta que Python distingue entre mayúsculas y minúsculas. Es decir, que A y a son para Python variables distintas.

Las letras permitidas son las del alfabeto inglés, por lo que están prohibidas la ñ, la ç o las vocales acentuadas. Las palabras reservadas del lenguaje también están prohibidas. En caso de que intentes dar un nombre incorrecto a una variable, Python mostrará un mensaje de error al ejecutar el programa.

Cada variable se identifica por su nombre, así que en principio no puede haber dos variables distintas con el mismo nombre. (La verdad es que sí que puede haber dos variables distintas con el mismo nombre, pero sólo si cada una de las variables "existe" en su propio espacio, separada de la otra. Lo veremos más adelante.)

Aunque no es obligatorio, normalmente conviene que el nombre de la variable esté relacionado con la información que se almacena en ella, para que sea más fácil entender el programa. Mientras estás escribiendo un programa, esto no parece muy importante, pero si consultas un programa que has escrito hace tiempo (o que ha escrito otra persona), te resultará mucho más fácil entender el programa si los nombres están bien elegidos. También se acostumbra a utilizar determinados nombres de variables en algunas ocasiones, como irás viendo más adelante, pero esto tampoco es obligatorio.

Al definir una cadena, es decir una variable que contiene letras, debes escribir su valor entre comillas (") o apóstrofos ('), como prefieras.

Tipos de variables

Como hemos visto en el ejemplo anterior tenemos definiciones de algunos tipos de variables que hay en Python:

- números decimales,
- números enteros,
- cadenas (una o más letras)
- listas (conjuntos de números, cadenas o listas).

Python es un lenguaje fuertemente tipado: No se puede tratar a una variable como si fuera de un tipo distinto al que tiene.

Aunque las definamos de forma similar, para Python no es lo mismo un número entero, un número decimal, una cadena o una lista ya que, por ejemplo, dos números se pueden multiplicar pero dos cadenas no (curiosamente, una cadena sí que se puede multiplicar por un número).

Por tanto, estas tres definiciones de variables no son equivalentes:

```
>>>Fecha = 2010
>>>Fecha = 2010.0
>>>Fecha = "2010"
```

En el primer caso "Fecha = 2010" la variable Fecha está almacenando un número entero, en el segundo Fecha "Fecha = 2010.0" está almacenando un número decimal y en el tercero Fecha "Fecha = "2010"" está almacenando una cadena de cuatro letras.

Averiguando el tipo

Para saber el tipo de una variable, tenemos la función **type**, que se usa del siguiente modo:

```
type(MiVariable)
```

Donde MiVariable es la variable de la que queremos saber el tipo.

Esta función retorna el tipo de dato de la variable.

Por ejemplo: Prueba a ejecutar el siguiente código:

```
#!/usr/bin/python
# coding: UTF-8
# Esto es un entero
Entero = 32

# Esto es una cadena
Cadena = "Hola Mundo"
print type(Entero)
print type(Cadena)
```

Existen una serie de "*alias*" previstos para poder hacer comparaciones con los valores que retorna type(), de los cuales los más usado son los siguientes:

- **bool** Para valores lógicos "Cierto" o "Falso", como se verá más adelante en este curso.
- **float** Para números en coma flotante.
- **int** Para números enteros.
- **str** Para variables de tipo cadena.

En realidad, type no sólo funciona con variables, si no que también tiene uso en contextos de cadenas, arrays, funciones, clases, módulos y cualquier otra estructura que pueda tener un tipo.

Manipulando tipos de datos

Dado que Python es tan exigente a la hora de tratar con los tipos de datos, se hace necesario un procedimiento para cambiar de un tipo a otro, para así poder operar con ellos.

Python posee una serie de funciones que permiten esto mismo, y que fincionan del mismo modo general: Se les introduce un dato de un tipo determinado y retornan ese mismo valor, pero de un tipo distinto.

Los más habituales son los siguientes:

- **str(MiVariable)** Convierte MiVariable a una cadena.
- **int(MiVariable)** Convierte MiVariable a un entero.
- **float(MiVariable)** Convierte MiVariable a un número en coma flotante.

El siguiente código cambiaría el tipo de dato de Entero (int) a Cadena (str):

```
#!/usr/bin/python
# coding: UTF-8

# Esto es un entero
Entero = 2000

#Convertimos a Cadena
Cadena= str(Entero)

print type(Entero)

print type( Cadena)
```

Definir varias variables a la vez

En una misma línea puedes definir simultáneamente varias variables, con el mismo valor o con valores distintos, como muestra el siguiente ejemplo:

```
>>> a = b = 69
>>> c, d, e = "Coche", 10, ["huevos", "patatas"]
```

Mostrar el valor de las variables en el IDLE

Para que el IDLE nos muestre el valor de una variable, sólo tenemos que escribir su nombre. También podemos conocer el valor de varias variables a la vez escribiéndolas entre comas (el IDLE nos las mostrara entre paréntesis).

```
>>> a, b, c = 69, 'coche', "A"
>>> a
69
>>> c, b ('A', 'coche')
```

Utilizar variables ya definidas

Una vez que tenemos definida una variable, la podemos usar para hacer cálculos o para definir

nuevas variables, veamos algunos ejemplos:

```
>>> a = 2
>>> a + 3
5
```

Si por casualidad usamos una variable que no este definida, Python nos informara con un mensaje de error. Recordemos que los nombres de las variables no tienen sentido real, pues Python las usa como simples etiquetas para referirse al contenido.

Veamos un ejemplo:

```
>>> higos = 12
>>> peras = 13
>>>higos + peras
25
```

También podemos definir una variable usando su propio valor:

```
>>> a = 10
>>> a = a+5
>>> a
15
```

Introduciendo datos

Las variables no tendrían sentido si no hubiese un modo de asignarles valor.

Se podría obligar al usaurio a editar el código cada vez que quisiese cambiar el valor de una variable pero, probablemente, resultaría poco cómodo.

Para permitir al usuario asignar un valor a una variable en la línea de comandos, existe la función "input".

Su uso es muy simple:

```
    MiVariable= input('Texto de Prompt')
```

En el ejemplo anterior, **MiVariable** es la variable a la que le queremos asignar un valor, y **"Texto de Prompt"** es una cadena, elegida por nosotros, que se mostrará al solicitar el valor.

Por ejemplo:

```
    Edad= input('Introduzca su edad:')
```

Mostraría en pantalla el texto "Introduzca su edad:" y esperaría a que el usuario introdujese el dato, que sería asignado a la variable "Edad".

Pero input(), de cara al usuario, tiene un problema que lo hace poco práctico.

La función "input" espera que se le introduzcan los valores formateados al estilo Python. de modo que para asignarle una cadena, esta debe ser introducida entre comillas, o dará un error.

Para resolver esto, existe la función raw_input, que se comporta exactamente igual que input, pero considera toda entrada como una cadena:

```
    Cadena= raw_input('Introduzca el texto que quiera:')
```

Esto solucina el problema de cara al usuario, pero añade la incomodidad al programador de tener que transformar luego los tipos.

Por qué usar raw_input

Al usar input en lugar de raw_input, el tratamiento de datos se hace más cómodo, porque el propio intérprete se ocupa de averiguar qué es una cadena, qué es un entero, etc. Pero la entrada de datos por parte del usuario se hace más incómoda, porque las cadenas de texto deben estar entrecomilladas para que python sepa que lo son. Si se introduce sin comillas, el intérprete creerá que es el nombre de una variable.

Esto, a su vez, plantea un problema de seguridad. Por ejemplo, en el siguiente código:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

clave= "secreto"

entrada= input('Introduce la clave ')

if entrada == clave:
       print "acertaste, la clave es '", entrada, ""
else:
       print "acceso no autorizado"
```

Si, en la entrada, escribimos "clave" (sin las comillas), python lo interpreta como el nombre de una variable, por lo que lo reemplaza con el contenido de la variable de ese nombre y, por tanto, lo da por bueno independientemente de cuál sea el valor de la clave realmente. Es un ejemplo bastante

simple, pero muestra claramente el peligro de permitir el acceso directo a las variables del programa.

Para eso se usa raw_input, que funciona igual que input pero asume, automáticamente, que toda entrada es una cadena.

Operadores aritméticos

Un operador no es más que un carácter o símbolo que indica al intérprete que debe efectuarse una operación sobre los datos a los que acompaña.

Cuando, en ejemplos anteriores, se ha visto el uso del signo "=" para asignar un valor a una variable, se estaba usando el "operador de asignación".

Los operadores más básicos son los aritméticos, que permiten tareas simples con números, tales como la suma o la resta:

Nombre	Símbolo	Ejemplo	Explicación
Suma	+	resultado= primera + segunda	Suma las cifras o variables
Resta	-	resultado= primera - segunda	Resta las cifras o variables
Multiplicación	*	resultado= primera * segunda	Multiplica las cifras o variables
División	/	resultado= primera / segunda	Divide las primera cifra o variable por la segunda
División entera	//	resultado= primera // segunda	Como el anterior, pero sin decimales
Módulo	%	resultado= primera % segunda	Obtiene el resto de divir la primera por la segunda
Exponente	**	resultado= primera ** segunda	Eleva la primera a la potencia de la segunda

En capítulos posteriores se verán otros tipos de operadores más avanzados, como los lógicos.

¿Que es una Lista?

La lista es una colección de datos ordenada, alguna equivalencia con otros lenguajes seria los arrays o vectores.

La lista puede contener cualquier tipo de dato (enteros, cadenas y otras listas)

Vamos a ver como se puede crear una lista:

```
>>>lista = [''hola'',69,''curso'',[1,2,3,4,5]]
```

Como observamos nuestra lista principal tiene un string "hola", un entero 69 y otra lista, y por ultimo una lista dentro de otra lista.

```
>>>print (lista)
['hola', 69, 'curso', [1, 2, 3, 4, 5]]
```

Movernos por la lista

Si queremos movernos por la lista y acceder a uno de los elementos de la lista lo hacemos utilizando el nombre al que hacemos referencia a la lista en este caso lo llame lista puede ser cualquiera y entre corchete indicamos un índice, dicho índice va de 0 a n-1

Vamos a ver un ejemplo:

Tenemos la lista anterior

```
lista = [''hola'',69,''curso'',[1,2,3,4,5]]
>>>print(lista[0])
hola
>>>print(lista[3])
[1, 2, 3, 4, 5]
```

Si queremos acceder al 2 dentro de la lista [1,2,3,4,5] es tan simple como:

```
>>>print(lista[3][1])
2
```

Como vimos con el operador [ ] podemos hacer referencia a cualquier elemento de la lista de 0 a n-1 pero Python trae consigo algo que es recorrer la lista de ultimo al primero utilizando números negativos veamos un ejemplo:

```
>>> nombres = [''Pablo'',''Fran'',''José Antonio'',''Juan Julián'']
>>>print (nombres[-1])
Juan Julián
>>>print (nombres[-4])
Pablo
```

Como siempre nos gusta poner a prueba el lenguaje; que nos pasa si ponemos, print(nombres[-5]) o print (nombres[4]) nos genera un error que dice, lista fuera de rango así:

```
Traceback (most recent call last):
File "", line 1, in
IndexError: list index out of range
```

Modificar una lista

Como modificamos el valor de una lista en concreto, pues es muy fácil, veámoslo, seguimos usando la lista anterior.

```
>>>lista[0]=''Python''
>>>print(lista)
['Phython', 69, 'curso', [1, 2, 3, 4, 5]]
```

Agregar y quitar valores

Metiendo Datos

La instrucción básica para añadir elementos a una lista es append(), que agrega un nuevo elemento al final de esta.

Su uso sería:

```
MiLista.append(NUevoElemento)
```

Donde **NUevoElemento** es lo que queremos agregar a la lista **MiLista**.

Este tipo de notación, con un elemento seguido de un punto y la función a aplicarle, es propia de los objetos y se verá con algo más de detalle en temas siguientes.

Como nota, para ia abriendo boca, agregaremos que en Python todo dato, función, estructura, etc, puede ser manipulado como un objeto.

También se puede introducir un dato "en medio" de la lista, usando insert() del siguiente modo:

```
MiLista.insert(Posicion,NuevoElemento)
```

Donde **Posicion** es el número de posición donde se insertará, y **NuevoElemento** el elemento introducido.

Es importante recordar que el nuevo elemento no reemplaza el contenido anterior, si no que el dato que anteriormente hubiera en esa posición se desplazará un paso más adelante.

Sacando Datos

La forma mas simple de eliminar datos de una lista es por medio de la función pop().

Esta función es muy útil, porque elimina el último elemento de una lista y lo retorna. De modo que se puede usar del siguiente modo:

```
ValorExtraido= MiLista.pop()
```

Esto eliminaría el último elemento de **MiLista** y lo asignaría a la variable **ValorExtraido**.

Intuitivamente, puede verse que el manejo básico de una lista como buffer es a base de append() y pop().

Particionar listas

Este mecanismo nos permite seleccionar porciones de las listas, vamos a conocer como funciona este mecanismo.

```
>>> nombres = [''Pablo'',''Fran'',''José Antonio'',''Juan Julián'']
>>>print(nombres[1:3])
['Pablo', 'José Antonio']
```

Hasta ahora es fácil, si queremos indicar porciones de lista damos un numero inicial en donde quiero que inicie mi parte seguido de (:) dos puntos y un numero final que me dice el limite de la lista.

CUIDADO: ese numero final actua o me imprime el elemento de la lista contenida en (numerofinal-1) por eso imprime José Antonio y no Juan Julián, si queremos que vaya hasta python solo es:

```
>>>print(nombres[1:4])
['Fran', 'José Antonio','Juan Julián']
```

Vamos a conocer mas sobre esto, creamos una lista de enteros:

```
>>>n=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29,30]
```

Tenemos una lista y vamos a usar la estructura inicio-fin-salto para que podamos extraer los elementos de la lista.

- solo con (inicio:fin)

```
>>>print(n[2:8])
[ 3, 4, 5, 6, 7, 8 ]
```

- ahora con (inicio:fin:salto)

```
>>>print(n[2:8:2])
[3, 5, 7 ]
```

Arrays asociativos o mapas

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor.

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas, … pero no listas o diccionarios, dado que son mutables. Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

Por ejemplo:

```
>>> mydict = {"altura" : "media", "habilidad" : "alta", "salario" : 999}
>>> print mydict
{"altura" : "media", "habilidad" : "alta", "salario" : 999}
>>> print mydict ["habilidad"]
alta
```

Arrays asociativos II

También podemos comprobar la existencia de una clave en un diccionario usando has_key:

```
if mydict.has_key ('altura'):
print 'Nodo encontrado'
```

También podríamos hacer :

```
if 'altura' in mydict:
print 'Nodo encontrado'
```

Estructuras de control básicas

Introducción

Un programa medianamente útil necesita reaccionar ante cambios en su "entorno", de modo que se ejecuten distintas secciones del código en distintas circunstancias. Para ello existen las estructuras condicionales.

Las estructuras condicionales son constructos de software que se encargan de comparar valores (ya sea de variables, constantes, arrays...) y, en función del resultado de esa comparación, ejecutar o no un determinado bloque de código.

Un bucle (ciclo, loop) es un segmento de código que se ejecuta repetidas veces mientras se cumpla una condición determinada.

Teorema de la programación estructurada

El teorema de la programación estructurada afirma (y demuestra) que toda función computable puede codificarse usando sólo combinaciones de tres tipos de estructuras básicas:

Secuencia:

Se ejecuta una orden y despues otra, secuencialmente, en lo que viene a ser el flujo "natural" de un programa.

Condicional:

Se ejecuta una de dos órdenes, dependiendo de un valor lógico boolenao ("cierto" o "falso"). Naturalmente, esto puede combinarse para multiplicar las diversas opciones, y unirse a lo anterior para elegir entre bloques completos de código.

Bucle:

Se ejecuta una orden mientras una condición sea cierta. O (lo que viene a ser lo mismo) se ejecuta hasta que deje de serlo.

Cualquier otra estructura (como, por ejemplo y sobre todo, el famoso -e infame- GOTO) es prescindible o sustituible por las anteriores y, en general, sólo contribuye a dificultar la legibilidad del código.

Respecto a esto último, el polémico y genial Edsger Dijkstra dijo del lenguaje BASIC y su uso extensivo de la sentencia GOTO que "*[...] la enseñanza de BASIC debería ser considerada un delito penal: Mutila la mente más allá de toda recuperación*".

Anidamiento

Las instrucciones pueden "anidarse". Es decir, pueden incluirse unas dentro de otras para hacer dependiente su ejecución.

En otros lenguajes de programación es habitual anidar las sentencias por medio de símbolos como:

```
Instruccion_Principal {

    instruccion_anidada

}

instruccion_independiente
```

En Python, el anidamiento de instrucciones no se hace con llaves ni otros símbolos contenedores, si no por medio de la indentación. las instrucciones con un nivel de indentación (digamos que un tabulador, ver más abajo) estań contenidas en las que no están indentadas que les preceden, las instrucciones con dos tabuladores están incluidas en las anteriores, etc.

```
Instruccion_Principal

    instruccion_anidada

instruccion_independiente
```

Se recomienda encarecidamente que no se use realmente el carácter tabulador (ASCII 09) para indentar, si no que se usen espacios. Lo recomendable, según Guido van Rossum, son cuatro espacios por cada nivel de indentación (Sin embargo, Linus Torvalds considera cualquier tabulación de menos de 8 caracteres como una herejía). Nunca mezcles indentados por espacios e indentados por tabulador.

Comentarios

Para ayudar a mantener el código, indicar detalles o, en general, añadir textos que deben ser leído por personas pero no por el interprete o el compilador, los lenguajes de programación suelen incluir algún modo de indicar que algo es un "comentario", y no una instrucción.

En Python, los comentarios comienzan con el signo almohadilla (#) seguido de un espacio.

La siguiente línea sería un comentario y, por lo tanto, sería ignorada por el intérprete de python:

```
# Esto es un comentario
```

Esto ocurre unque lo que haya tras el # sea código de python. En el siguiente ejemplo el valor final de "variable" es 10, y no 5:

```
variable= 10
# variable= 5
```

Un comentario también puede estar a continuación de código interpretable, en la misma línea, de este modo:

```
gravedad= 9.8     # aceleración de gravedad en la Tierra, en m/s^2
```

Es más que recomendable usar los comentarios para documentar el código lo máximo posible, tanto para los demás como para leerlo tú mismo. Hoy puedes saber qué

hace exactamente, pero mañana lo olvidarás.

En palabras de Martin Golding: "*Programa siempre como si el tipo que acabe manteniendo tu código fuera un psicópata violento que sabe dónde vives.*"

Además, ṕracticamente todo programa debería empezar con un comentario indicando qué es, para qué se usa, el autor y la licencia:

```
##########################################
#
# Copyright 2011 Allan Psicobyte
#
# Programa para resolver problemas NP en tiempo polinómico
#
# Es software libre y se distribuye bajo una licencia Affero (AFFERO
GENERAL PUBLIC LICENSE: http://www.affero.org/oagpl.html).
#
# This program is free software and it's licensed under the AFFERO GENERAL
PUBLIC LICENSE (http://www.affero.org/oagpl.html).
##########################################
```

¿Te ha sabido a poco? Un método más avanzado de mantener una buena documentación en Python

es por medio de Docstrings.

Cierto y falso

Las estructuras que permiten controlar el flujo de un programa se basan, fundamentalmente, en el "valor de verdad" de una sentencia.

¿Qué es un valor lógico "cierto" o "falso" para Python?

En general, los valores no definidos, cadena vacía, lista vacía y el 0 son interpretado como "falso".

El resto de valores se consideran "ciertos".

Naturalmente, el valor de verdad puede ser (y suele ser) el resultado de una operación de comparación.

Los operadores de comparación son los siguientes:

- menor que ("<")
- mayor que (">")
- igual a ("==")
- menor o igual que ("<=")
- mayor o igual que (">=")
- distinto de ("!=")
- está en ("in")
- no está en ("not in")
- es ("is")
- no es ("is not")

Todos estos operadores retornarán valores de "cierto" o "falso".

Se pueden unir proposiciones con los habituales operadores lógicos "and" (y) y "or" (o), y también existe la preposición "not" (no), para invertir el valor de verdad.

Condicional

En Python, el anidamiento de funciones no se hace con llaves ni otros símbolos contenedores, si no por medio de la indentación. las instrucciones con un nivel de indentación (un tabulador) estań contenidas en las que no están indentadas que les preceden, las instrucciones con dos tabuladores están incluidas en las anteriores, etc.

La estructura básica para el control del flujo de un programa es el "si condicional", que **permite** la

ejecución de un segmento de código dependiendo de las condiciones concretas.

```
If expresion_a_evaluar:

    ejecutar_si_cierto
```

(Nota los dos puntos (**:**) detrás de la expresión)

En este caso, la sentencia "ejecutar_si_cierto" solo se ejecutará si "expresion_a_evaluar" devuelve un valor lógico "cierto", y se ignorará si retorna un valor de "falso".

La anterior estructura puede ampliarse añadiendo un bloque alternativo:

```
if expresion_a_evaluar:

    ejecutar_si_cierto

else:

    ejecutar_en_caso_contrario
```

En este caso, el código incluído dentro del bloque **else** se ejecutará sólo si "expresion_a_evaluar" tiene un valor falso.

O también existe la siguiente generlización:

```
if expresion_a_evaluar_1:

    ejecutar_si_cierto_1

elif expresion_a_evaluar_2:

    ejecutar_si_cierto_2

elif expresion_a_evaluar_3:

    ejecutar_si_cierto_3

else:

    ejecutar_si_ninguna
```

En la que se dan varias opciones, que se ejecutarán cuando sea cierta la expresión correspondiente, y una última que sólo se ejecutará si no es cierta ninguna de las anteriores.

Por ejemplo:

```
if x > 10:

    print "La variable es mayor que diez"

elif x < 10:


    print "La variable es menor que diez"

else:

    print "La variable es, precisamente, diez"
```

Bucle while

la instrucción while crea un blucle que se ejecutará mientras su condición sea cierta.

La estructura de while es la siguiente:

```
while condicion:
 instruccion_a_ejecutar
```

Por ejemplo:

```
while a < 10:
    print a
    a = a + 1
```

Este ejemplo irá recorriendo el bucle, imprimiendo el valor de "a" y sumándole uno, mientas este sea menor que 10.

While también permite una sentencia "else" que se ejecutará cuando la condición del blcle no sea cierta (es decir, fuera del blucle) de este modo:

```
while condicion:

    instruccion_a_ejecutar

else

    instruccion_fuera_del_bucle
```

Bucle for

El bucle for sirve para recorrer secuencialmente los elementos de una lista.

La estructura de for es la siguiente:

```
for Variable in Lista
    instrucciones
```

Esto ejecutaría "instrucciones" para cada elemento de "Lista", que se ha ido almacenando en "Variable". Veamoslo con un ejemplo:

```
Huerto = ["zanahoria", "col", "lechuga", "col"]
for Planta in Huerto:
    if Planta != "col"
        print Planta
    else
        print "Odio las coles"
```

Este ejemplo imprimirá cada uno de los valores de la lista "Huerto", a menos que este sea "col", en cuyo caso imprimirá el texto "Odio las coles".

El bucle for, al igual que vimos con while, también permite el uso de una cláusula "else" que se ejecutará cuando no quede ningún valor en la lista.

```
for Variable in Lista
    instrucciones
else
   ya_no_quedan_elementos
```

A veces, este tipo de bucle puede resultar poco intuitivo. Aquellos que estén acostumbrados a otros lenguajes de programación comprobarán que el uso de *for* en Python (como una herramienta de recorrido de arrays) se corresponde, a lo que en lenguajes como Perl se llamaría foreach, mas que a un for típico.

Break y continue

Adicionalmente al control que nos permiten las instrucciones for y while, Python nos ofrece herramientas para ajustar su comportamiento.

La sentencia break, dentro de for o while, permite interrumpir el flujo normal del bucle, saliendo

automáticamente de él independientemente de si se cumple o no la condición del bucle.

Por ejemplo:

```
x= 0
while x < 10
    if x == 5
        break
    print x
    x = x + 1
```

Este bucle se interrumpirá cuando x valga 5 (valor que no llegará a imprimirse), aunque se siga cumpliendo la condición de while.

Hay que precisar que break sale **completamente** del bucle, con lo que no se ejecutará tampoco ninguna instrucción "else" que este pudiera tener.

Menos "dramática" que break es la instrucción "continue". Esta permite interrumpir el ciclo como lo hace break, pero sin salir del bucle. De este modo sólo se interrumpe una iteración.

Por ejemplo:

```
x= 0
while x < 9
    x = x + 1
    if x == 5
        continue
    print x
```

Este ejemplo, pese a ser casi idéntico al anterior (sólo se ha reemplazado el "break" por un "continue") tiene un resultado distinto. Esta vez se imprimirán todos los números del 1 al 10 (como está especificado en la cláusula del while), pero omitiendo el 5 (que ha sido saltado por el "continue").

Control de excepciones

Cosas como la clásica división por cero o el tratamiento de tipos de datos incompatibles (sumar cadenas, por ejemplo) provocarán errores en tiempo de ejecución (excepciones) que darán al traste con el programa.

Para facilitar el manejo de este tipo de cosas tenemos la estructura *try:*

Dicho de un modo simple, lo que hace *try* es ejecutar un bloque de sentencias en un "entorno controlado", para que el error generado (si se da) no detenga el programa, si no que se retorne de modo que pueda manejarse.

Veamoslo con un ejemplo. En el siguiente bloque:

```
resultado = dividendo/divisor
print "La división resulta: ", resultado
```

Si *divisor* tuviese el valor "0" el programa daría un error y se interrumpiría. Para prever esa posibilidad se puede modificar así:

```
try:
        resultado = dividendo/divisor
        print "La división resulta: ", resultado

except:

        if divisor == 0:
                print "No puedes dividir por cero, animal"
```

El bloque dentro de *try* es ejecutado y **si retorna cualquier error**, entonces ejecuta el bloque contenido en *except*. en caso contrario se continúa la ejecución del programa ignorando ese bloque.

(De acuerdo, la descrita es una solución muy limitada, pero se puede incluir en un bucle mayor para que permita volver a introducir el dato, etc...)

Dado que existen muchos tipos de erro distintos, sería deseable una forma algo más sofisticada y concreta de manejar estos. en el caso de arriba, por ejemplo, el programa puede dar un error si divisor es cero o si es una cadena, y sería deseable manejar ambos casos de distinta manera.

Para ello, *except* permite escribirse de modo que indique le tipo de error concreto al que responde, de la siguiente forma:

```
except Tipo_de_Error:
```

Y, además, pueden colocarse tantos bloques except como sean necesarios.

De este modo, nuestro ejemplo se podría mejorar mas o menos así:

```
try:
        resultado = dividendo/divisor
        print "La divisón resulta: ", resultado

except ZeroDivisionError:

        if divisor == 0:
                print "No puedes dividir por cero, animal"

except ValueError:

        if divisor == 0:
                print "Hay que ser bruto: eso no es un número"
```

Cada uno de los bloques *except* se ejecuta sólo si se da el tipo de erro especificado.

Puedes ver un listado exahustivo de códigos de excepciones y su descripción en la documentación de python.

Si estás pensando que esto se parece a un if un tanto sofisticado tienes razón. De hecho, esta estructura permite también una sentencia else, que se ejecuta cuando no hay errores. De este modo, el ejemplo anterior sería más correcto escrito así:

```
try:
        resultado = dividendo/divisor

except ZeroDivisionError:

        if divisor == 0:
                print "No puedes dividir por cero, animal"

except ValueError:

        if divisor == 0:
                print "Hay que ser bruto: eso no es un número"
else:
        print "La divisón resulta: ", resultado
```

Nota que dentro del try hemos dejado sólo la instrucción que requiere que verifiquemos, dejando el "print" en el else final.

Funciones y Programación dirigida a objetos

Funciones, Funciones, Funciones

Antes de entrar en detalles, veamos una breve definición. Las funciones son el mecanismo universal de la programación estructurada. Si has trabajado con otros lenguajes de programación puedes identificarlas como subrutinas o procedimentos. Las funciones tienen dos roles principales en el desarrollo de software:

a) Maximizar la reutilización y minimizar la redundancia de código.

b) Descomposición Procedimental

Redundancia

Como en la mayoría de los lenguajes de programación, las funciones de Python son la vía mas simple para empaquetar un concepto lógico, te permiten agrupar y generalizar código fuente que usaras varias veces a lo largo del código de tu software.

Las funciones son la herramienta básica en la factorizacion de código fuente: te ayudara a eliminar el código redundante en tus programas, y como consecuencia reducirás el posterior mantenimiento del mismo.



Descomposición Procedimental

Las funciones como herramienta, te permiten dividir los programas en piezas que tienen muy bien definido su rol.

Por ejemplo, si quieres programar un robot para hacer pizza, mediante las funciones podrás dividir la tarea principal "Hacer pizza" en varios trozos/subtareas, en la que cada subtarea sera una función: hacer base, añadir ingredientes, meter en el horno.

Es mas sencillo de implementar pequeñas subtareas, que implementar todo el proceso de golpe.

Creando Funciones

Otra forma de definir el concepto de función es como un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None, equivalente al null de Java.

El formato general para usar en funciones Python, es el siguiente:

```
def (arg1,arg2,... argN):
```

Aparece la palabra reservada def, la cual crea una función y le asigna un nombre. En las siguientes lineas, indentadas, se coloca el cuerpo de la función. Estas instrucciones se ejecutaran cuando la función sea llamada.

A continuación del nombre, como en la mayoría de lenguajes, aparece la lista de argumentos, que puede estar vacía.

Cuando sea necesario en el cuerpo de la función se utizara la palabra reservada return:

```
def (arg1,arg2,... argN):
….
return
```

Se utilizar al final de cuerpo, para enviar un resultado al lugar donde se hizo la llamada a la función.

Veamos un primer ejemplo en la siguiente imagen:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def tablasMultiplicar():
    for i in range(0,11):
        print "\n Tabla del ", i , "\n"
        for j in range(0,11):
            print i, " X ", j, " = ", i*j


tablasMultiplicar()
```

Paso de argumentos

En el ejemplo del apartado anterior se ha declarado una función que imprime por pantalla las 10 primeras tablas de multiplicar. Mediante el paso de argumentos podemos exprimir nuestra función. Imagina que solo necesitas imprimir la tabla de multiplicar de un determinado número. Se escribe el identificador de nuestro parámetro dentro de los paréntesis en la declaración de la función. Sintaxis:

def tablasMultiplicar(numero):

Este parámetro lo usaremos en el cuerpo de la función para resolver nuestro problema. Se propone como ejercicio practico.

Si necesitamos varios argumentos los separaremos mediante ','.

A una función se le pueden pasar tantos argumentos como necesitemos, pero si tu función tiene mas de 5 o 6 argumentos, piensa en resolver tu problema de otra forma y hacer que tu código sea mas "legible".

Como alternativa podrás usar una lista donde almacenar los parámetros, y que esta lista sea el argumento de tu función.

Una función especial

Al igual que la palabra reservada def, Python tiene una herramienta para crear funciones. Es un objeto similar al que aparece en el lenguaje de programación Lisp, por este motivo se llama lambda.

Son funciones que no tienen nombre/identificador. Sencillamente reciben una lista de argumentos, y devuelven un valor.

Sintaxis

```
lambda arg1,arg2,... argN:
```

Ejemplo de uso:

```
f = lambda x,y,z: x+y+z //declaracion
f(2,3,4) //llamada
9 //Resultado
```

La expresión lambda es muy útil cuando tienes que usar pequeños trozos de código ejecutable donde las palabras reservadas como por ejemplo def son incorrectas sintácticamente.

Veamos un ejemplo (jump tables) en el siguiente recurso.

Ejemplo de uso: Lambda

```
L = [lambda x : x ** 2,
    lambda x : x ** 3,
    lambda x : x ** 4]

for f in L:
    print(f(2))
        #Resultado 4,8,16

print(L[0](3))
    #Resultado 9
```

Un ejemplo mas

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

#Libreria para generar numeros aleatorios
import random

def numeroAleatorio():

    #Numero entero aleatorio entre 0 y 100
    return random.randint(0,100)


variable = numeroAleatorio()

print "Contendio de mi variable", variable
print "Valor devuelto por una funcion", numeroAleatorio()
```

Orientación a Objetos

Python es un lenguaje de programación orientado a objetos. Por lo general el paradigma orientado a objetos resulta adecuado cuando queremos agrupar estados (datos) y comportamiento (código) juntos en prácticos paquetes de funcionalidad.

En los apartados anteriores se han mostrado formas de organizar los datos y el código. Una lista nos permite utilizar variables(datos), y una función permite almacenar bloques de código, de tal forma que se podrá usar una y otra vez. Mediante el uso de objetos agruparemos variables y funciones.

A continuación hay un recurso que contiene información detallada sobre el paradigma de la orientación a objetos. Si no lo has aplicado en otros lenguajes de programación, lee detenidamente el recurso.

http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

Clases e Instancias

Si ya estamos familiarizados con la orientación a objetos en lenguajes como Java o C++, con toda probabilidad tendrás un buen dominio de las clases e instancias:

Una clase es un tipo de dato definido por el usuario, que podemos ejemplificar(instanciar) para obtener instancias, es decir, objetos de ese tipo. Python admite estos conceptos mediante sus objetos de clase e instancia.

Características de una clase:

- Podemos llamar a un objeto de clase como si fuera una función. La llamada devuelve otro objeto, denominado instancia de la clase. La clase también se conoce como el tipo de la instancia.
- Una clase tiene atributos nominales que podemos vincular y hacer referencia.
- Los atributos de clase vinculados a funciones también se conocen como métodos de la clase.
- Una clase puede heredar de otras clases, es decir, delega a otros objetos de clase la búsqueda de los atributos que no se encuentran en esa clase.

Una instancia de clase es un objeto de Python con atributos nominales los cuales podemos vincular o hacer referencia. En Python las clases son objetos (valores) y se manipulan como otros objetos.

La sentencia class

La sentencia class es el modo habitual de crear un objeto de clase. class es una sentencia compuesta de clausula única, con la sintaxis:

```
class nombre_clase():
atributos
métodos
```

El cuerpo de la clase es donde especificamos por lo general los atributos de la clase; estos atributos pueden ser objetos (incluyendo las funciones) u objetos de datos normales: entero, cadena, lista, tupla...

Para declarar datos normales, escribimos el identificador del dato y su correspondiente valor, por ejemplo:

```
class c1():
x= 25
```

En el caso de los métodos es necesario el uso de la sentencia def:

```
class c1():
     x=25
     def doble(self):
     return x * 2
```

En el ejemplo final, aparece un método especial __init__ se le conoce en el ámbito de la programación orientada a objetos como el constructor de la clase. Con este método creamos un constructor del objeto Mensaje, es decir, este método crea un objeto de tipo Mensaje y le asigna a su atributo texto el valor que se le pasa como parámetro, cadena.

Ejemplo de declaración y uso de una clase sencilla.

```
class Mensaje:
    #contructor
    def __init__(self, cadena):
        self.texto = cadena
    def printMensaje(self):
        print self.texto

#Hola Mundo escrito en Criollo Haitiano
m1 = Mensaje("Hello Mondyal!")

m1.printMensaje()
```

'self' o no 'self'

Python es explícito. Por lo tanto los métodos de la clase en Python aceptan un parámetro adicional,se sitúa en primer lugar y es la instancia de la clase. Este parámetro es 'self'. Es idéntico a la variable $this en las clases de PHP y el uso de this en Java (aunque en Java, sólo se requiere para eliminar la ambigüedad).

```
#!/usr/bin/python

class Complejo:

    def __init__(self, parteReal, parteImaginaria):
        """Constructor de la clase Complejo"""
        self.r = parteReal
        self.i = parteImaginaria

    def getReal(self):
        return self.r

    def getImag(self):
        return self.i

num = Complejo(3.0,-4.5)
print "Parte Real ", num.getReal()
print "Parte Imaginaria ", num.getImag()
```

Ejemplo de creación de una clase, se define el objeto numero complejo con sus atributos y métodos. Destacar el uso del método __init__ como constructor de la clase, el cual recibe tres parámetros: self, parte imaginaria y real, para construir un objeto numero real.

PyRencia

Otra característica que tiene que tener un lenguaje para considerarse orientado a objetos es la HERENCIA.

La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.

En Python, a diferencia de otros lenguajes orientados a objetos (Java, C#), una clase puede derivar de varias clases, es decir, Python permite la herencia múltiple.

Superclase

Clase de la que desciende o deriva una clase. Las clases hijas (descendientes) heredan (incorporan) automáticamente los atributos y métodos de la la clase padre.

Subclase

Clase descendiente de otra. Hereda automáticamente los atributos y métodos de su superclase. Es una especialización de otra clase. Admiten la definición de nuevos atributos y métodos para aumentar la especialización de la clase.

En el siguiente ejemplo la superclase es Motor y las Subclases son Coche y Motocicleta que heredan los atributos de la clase Motor.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

class Motor :
    """Clase de la cual heredan las demas"""

    def __init__(self, marca) :
        self.marca = marca

    def imprimirMarca(self) :
        print "La marca es", self.marca

class Coche(Motor) :
    """Esta clase hereda de Motor"""
    pass

class Motocicleta(Motor) :
    """Esta clase hereda de Motor, pero es diferente a Coche"""
    pass


buga = Coche("Toyota")
moto = Motocicleta("Kawasaki")
buga.imprimirMarca()
moto.imprimirMarca()
```

PyMorfeando

Es muy simple, el polimorfismo viene a significar que puedes tener dos (o mas) métodos con el mismo nombre para diferentes clases. Estos métodos pueden comportarse de diferente manera dependiendo de la clase en que se apliquen.

Por ejemplo, para practicar la geometría decides escribir un programa que calcule el área de

diferentes figuras, como triángulos, rectángulos.

Para el ejemplo se crean dos clases y las dos tienen el mismo método getArea() pero se implementa de forma diferente paca cada clase. Esto es polimorfismo.

La siguiente imagen muestra un ejemplo de polimorfismo.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

class Triangulo:
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def getArea(self):
        area = self.ancho * self.alto / 2.0
        return area

class Rectangulo:
    def __init__(self, lado):
        self.lado = lado

    def getArea(self):
        area = self.lado * self.lado
        return area


miTriangulo = Triangulo(3,6)
miRectangulo = Rectangulo(5)

print miTriangulo.getArea()
print miRectangulo.getArea()
```

Uso de Librerías Estándar

El concepto de Módulo

Un modulo es una parte de algo. Algo es modular si es posible separarlo en partes o piezas. Los bloques de LEGO son el ejemplo perfecto de modularidad. Puedes coger diferentes piezas y crear

diferentes cosas con ellas.

En Python, un modulo es una pequeña pieza de un gran programa. Este modulo es un fichero dentro de tu disco duro.

Motivos por los que usar módulos:

* Se crean ficheros pequeños, donde es mas fácil de localizar elementos de tu código.

* Un modulo se puede usar en todos los programas que desees. Sin necesidad de repetir código fuente.

Para un nuevo modulo se crea un fichero y se nombra como mi_modulo.py en este fichero incluimos el código que necesitemos, por ejemplo funciones.

Para utilizar las funciones de nuestro modulo usaremos import mi_modulo, ya tendremos disponibles todas las funciones que añadimos a nuestro módulo, las cuales usaremos en el nuevo programa.

A la hora de importar un módulo Python recorre todos los directorios indicados en la variable de entorno PYTHONPATH en busca de un archivo con el nombre adecuado. El valor de la variable PYTHONPATH se puede consultar desde Python mediante sys.path

En los siguientes apartados conoceremos los módulos estándar de Python y las funciones mas utilizadas dentro de cada modulo.

Obtener Ayuda sobre un Módulo

A continuación practicaras con comandos que ofrecen información necesaria para hacer un buen uso de los módulos de Python

El comando dir(nombre_modulo) imprime un listado con las funciones disponibles para ese módulo.

Para obtener información especifica de una función incluida en un modulo, lanzaremos el comando help(nombre_modulo.nombre_funcion).

Prueba las siguientes ordenes en el interprete de Python:

```
>>> import os
>>> dir(os)
>>> help(os)
>>> help(os.getcwd)
```

Tareas del Sistema

Este módulo proporciona acceso a algunas variables utilizadas o mantenidas por el intérprete y con las funciones que interactúan fuertemente con el intérprete. Está siempre disponible.

De este modulo destacamos la función sys.path() es una lista de cadenas que especifica la ruta de búsqueda para los módulos

Interactuar con el Sistema Operativo

Este módulo proporciona una forma portátil de sistema operativo utilizando la funcionalidad de su cargo. Si lo que quiere es leer o escribir un archivo, ver open (), si quiere manipular rutas, consulte el módulo os.path, y si quieres leer todas las líneas en todos los archivos en la línea de comandos ver el módulo fileinput().

Para la creación de archivos y directorios temporales consulta el módulo de archivo temporal, y por el elevado nivel de archivo y manejo de directorio de ver el módulo shutil.

Manipular Cadenas de Caracteres

Para crear una cadena, escribe entre comillas. Las cadenas en Python se puede definir con comillas simples (') o comillas (").

La función len () devuelve la longitud de la cadena, es decir, el número de caracteres. Esta es la misma función que se utiliza para encontrar la longitud de una lista, tupla, conjunto, o un diccionario. Una cadena es como una tupla de caracteres.

Se pueden obtener caracteres individuales de una cadena usando la notación de índice.

Al igual que las listas, se pueden concatenar cadenas con el operador +.

Veamos unos ejemplos de uso en el siguiente recurso.

La librería String contiene una serie de constantes útiles y clases, así como algunas funciones que también están disponibles como métodos en cadenas.

Métodos

int len(string): retorna la longitud de una cadena.

int count(sub [,start[,end]]): retorna el número de ocurrencias de una cadena dentro de otra

boolean endswidth(sub [,start[,end]]): retorna verdadero o falso si una cadena termina con la cadena especificada.

boolean startswidth(sub [,start[,end]]): similar a endswidth.

int find(sub [,start[,end]]): retorna la posición numérica de la primera ocurrencia de una cadena dentro de otra

int rfind(sub [,start[,end]]): similar a find, pero en vez de retornar la posición de la primera ocurrencia, lo hace de la última

string lower(): retorna la cadena original en letras minúsculas.string upper(): retorna la cadena original en letras mayúsculas.

string replace(old, new [,count]): reemplaza todas o un número dado de ocurrencias de una cadena dentro de otra.

string strip([chars]): retorna una cadena eliminando al inicio y el fin de la misma un caracter dado. Si no se indica caracter utiliza como predeterminado el espacio en blanco.

array split([sep [,maxsplit]]): divide una cadena en un array de cadenas dado un separador. Si no se indica separador utiliza como predeterminado el espacio en blanco. Opcionalmente, se puede especificar el número máximo de divisiones.

string join(array): combina los elementos de un array de cadenas con otra cadena.

Uso de metodos

```
# -*- coding: utf-8 -*-

cad = "Hola Mundo !"

print cad[2]
print cad[2:5]
print cad[6:]
print cad[:9]

print len(cad)
print cad.count("o")
print cad.endswith("!")
print cad.find("M")
print cad.rfind("o")
print cad.upper()
print cad.replace(" ","_")

tupla = ["Carlos","Lopez", "Díaz"]
c= ","
print c.join(tupla)
```

Recuperar datos a partir de una URL

El modulo urllib2 puede leer datos de una URL usando varios protocolos como HTTP, HTTPS, FTP, o Gopher.

En el siguiente apartado aparece un ejemplo, en el que obtenemos el código html de un web.

```
#!/usr/bin/python

import urllib2

response = urllib2.urlopen('http://www.ugr.es/')
html = response.read()

print html
```

Veamos unos ejemplo ilustrativos sobre el uso de esta librería. ¿Esta disponible en la web de gomiso el episodio 23 de la seria The big bang theory ?, ¿La web de google contiene alguna imagen?, ¿Cuales son los 10 primeros caracteres de python.org?

Una vez que recuperamos el código html a partir de una url, podemos obtener una respuesta rápida a estas preguntas, por ejemplo, mediante el uso de métodos para el tratamiento de cadenas. Observa el siguiente bloque de código.

```
#!/usr/bin/env python

import urllib2

def search(url, cad):
    """Busca la cadena cad dentro del codigo html que contiene url """
    source = urllib2.urlopen(url).read()
    if source.find(cad) != -1:
        print "%s apararece en la web %s" % (cad,url)
    else:
        print "%s NO apararece en la web %s" % (cad,url)

if __name__ == '__main__':
    search('http://www.google.es','google')
```

El método search recupera el código html de la web indicada en el parametro url. En segundo lugar comprueba si la variable source contiene la cadena cad, que se pasa como parametro a la función. Por ultimo imprime un mensaje indicando el resultado de la búsqueda.

Para responder a la pregunta sobre una imagen en la web, simplemente la cadena a buscar seria el tag html para imágenes. En este caso en concreto seria img

Aumentar la funcionalidad de Python

Una de las ventajas de Python es la cantidad de módulos que se le puede instalar para aumentar su funcionalidad. Existen varias formas para poder obtener librerías adicionales y extender la funcionalidad de Python de acuerdo a nuestras necesidades.

Por ejemplo, PyPI es una página que tiene una recopilación de paquetes con múltiples propósitos listos para descargarse e instalarse.

Para poder administrar la instalación de todos estos paquetes existe pip que por medio de la consola nos facilita enormemente el trabajo de obtener e instalar cualquier paquete que lleguemos a necesitar.

Para instalar:

```
sudo apt-get install python-pip python-dev build-essential
sudo pip install --upgrade pip
```

Una vez realizado esto podemos instalar el paquete que necesitemos ingresando una simple instrucción. Por ejemplo, para instalar un módulo, usaríamos la instrucción:

sudo pip install nombre_modulo

pip se diferencia de otras herramientas parecidas como easy_install ya que primero descarga el paquete y luego lo instala por lo evitamos tener problemas si en algún momento la conexión a Internet se corta.

Otro modulo muy util para ver los paquetes instalados y disponibles es yolk, para instalarlo lanzamos en el terminal la siguiente orden: sudo install pip yolk

Para usarlo ejecutamos desde el terminal el comando yolk -l

Con un simple cauce podemos comprobar si tenemos disponible algún módulo. Prueba a lanzar esta orden desde el terminal: yolk -l | grep yolk